Digitális rendszer alapok és beágyazott C programozás

Forrás link: https://github.com/Benceking24/SEM_Prog_Alapozo

Mikroprocesszorok lényege:

- Kap egy külső adat forrást
- Feldolgozza kódolt logikat alapján
- Megjeleníti annak a következményét

Előnyük: Gyors, pontos, és olcsó + környezet specifikus (kevés áramfogyasztás, sok i/o, illesztett periféria)
Hátrányuk: Kis memória miatt azt nem éri meg dinamikusan kezelni + nincs védelem mint egy oprdenszerbe (igazából nem is probléma, mivel nem foglalkozunk több dologgal egy eszközön mint egy PC-n), nem stabil feszültség órajelingadozást okoz (ezért nehéz pontos RTC-t építeni velük)

Kódolás komplexitása: CircuitPython (**intepretálás** és compileolás) > Arduino Core > AVR C > Assembly
Egyéb opciók: WinAVR (“Whenever”) windowsra, CrossPack MacOSre, GCC-AVR debian/ubuntura.
Ezt követi a fordított és optimalizált gépi kód. Mivel x86 közeli és esetleg ARM architektúrájú gépekről fejlesztünk egy 8bites RISC architektúrára így a fordító egyben a **cross-compiler** szerepét is végzi.
Ezt a kódot az AVRdude vagy hozzá hasonló kód könyvtárak égetik a ROM és EEPROM-ba az ISP-n keresztül.

Fejlesztő környezetek: Arduino IDE (Processing könyvtár), AVR Studio, VsCode (Arduino/PlatformIO plugin) (Komplexebb környezet előnye: gépelés segítő **- kiegészítés**, forrás visszakeresése például függvényeknél - **navigáció**, debugolás például memória olvasás vagy soros port kommunikáció és plotter - **debug**, csoportos átnevezés - **refactoring**, integrált **verzió követő**)

Miért éri meg regisztereket piszkálni könyvtárak használata helyett? Ezek mind absztrakciós réteg (Arduino > Regiszterek > Assembly > gépi kód). Minnél lejebb vagy annál hatékonyabb lesz a kódod tehát több elfér belőle vagy gyorsabb lesz de egyben egyre szűkebb lesz a támogatott eszközök száma. Minnél feljebb vagy annál több mindenen fut ugyanaz a kód és kevesebb kódolás szükséges cserébe bölcsen kell minden utasítást használni.

ATmega32U4

Hasonló mint az Atmega328 (ami az ArduinoUno-ba van). Azért ez mert ugyanúgy futnak ugyanazok az utasítások ezen is, csak most ez volt kéznél egy Sparkfun Pro Micro formájában.

A **lábak** ‘Bank’-okba vannak csoportosítva mint több egyéb funkció is (mint pl.: external interrupt) és a 8-bites mivoltából 0-7es indexű egyedek vannak ezekbe. Egy ilyen láb több funkciót is el tud látni (ADC/Interrupt/Timer). Amikor resetelődik a chip ezek a lábak mind tri-state módba váltanak emiatt meg kell milyen célra szeretnénk őket használni (DDRD regiszter).

Programozás ISP-n keresztül. In-System Programming (3x2 vagy 5x2 tű). Úgy működik, hogy amikor áramot kap a chip és elkezdi olvasni a memóriát eléri a bootloader kódrészletet. Ez pedig arra figyel, hogy az RX lábon érkezik-e programozói utasítás és ha igen elkezdi másolni azokat a saját memóriájába. specifikáció:

http://ww1.microchip.com/downloads/en/appnotes/atmel-0943-in-system-programming_applicationnote_avr910.pdf

Szent grál: https://www.nongnu.org/avr-libc/user-manual/ (AVR-libC) alap C és AVR specifikus könyvtár. Pl.: Interrupt sei() és cli() vagy avr/io-ba a fejlesztendő chip leírása vagy stdio-ba fget ...stb

Bit műveletek:

Szimbólum	Logikai funkciója	Megjegyzés
| (altgr + w)	Vagy	bármelyik 1 akkor a kimenet is 1
&	És	mindkettő 1 akkor a kiement is 1
~	Negálás	megfordítja a bemenetet (~1 = 0)
^	Kizáró Vagy	akkor igaz ha CSAK az egyik bemenet igaz
<<	Balra eltolás	1<<0 = 0b0001, 1<<4 = 0b1000
>>	Jobbra eltolás	0b1000>>2 = 0b0010

C rövidítések:

Eredeti	Rövidített
a = a + 1	a += 1
a = a >> 1	a >>= 1
a = a & mask	a &= mask
a = a | mask	a |= mask
PORTB = PORTB ^ (1 << 0)	PORTB ^= (1 << 0)
if (feltétel) { utasítás ha igaz } else { ha hamis }	(feltétel) ? ha igaz: ha hamis ;

Gyakorlatban:

- n-edik bit 1-re állítása: *változó |= 1 << n;*
- n-edik bit 0-ra állítása: *változó &= ~(1 << n);*
- n-edik bit átbillentése: változó ^= 1 << n;

Interrupt: Egymagos tulajdonsága miatt lineáris munkavégzést képes folytatni az MCU. Ezt Belső vagy külső okok miatt meg tudjuk szakítani, hogy előre vegyük a sorba fontosabb/kisebb feladatokat. Az AVR világba az **ISR** (Interrupt Service Routine) végzi a vizsgálatot, hogy bekövetkezett-e egy ilyen interrupt és ha igen akkor melyik. Ezt követően a kódunkba külön megjelölt részt futtatja le. Azt hogy melyik interrupt állította meg a folyamatokat a “vector”-ok jelölik. (https://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html) Használatokor oda kell figyelni a következőkre:

- Rövid legyen
- Ne használj blokkoló utasítást (pl.: delay())
- Ne használj benne soros portra írást
- A benne használt változókhoz amit a *main* kód is használ írj ***volatile*** jelzőt

Végtelen ciklusra azért van szükségünk, hogy csináljon is valamit a chip még ha interrupt is van!

Pergés mentesítési módszerek:

- Hardware:
  - SR flip-flop
  - RC hálózat
- Szoftver:
  - Polling
  - Számlálás

Bővebben: https://my.eng.utah.edu/~cs5780/debouncing.pdf

További opciók és tulajdonságai

Biztosítékok: Ebbe vannak alapvető nem felülírandó hardware tulajdonságok leírva meg néha van hely fenntartva saját azonosító/kulcs beírására

Lock bit: Felprogramozható úgy hogy ne lehessen többet kiolvasni a chipet (lopásvédelem)

EEPROM memória: Extra adatot lehet tárolni benne ami nem kell sűrűn és szükséges, hogy újraindítások közt megmaradjon. Pl.: mérés adatok, digitális kulcsok, kép adat, hosszú szövegek
Időzítők: Az említett 8-16 bites időzítőn kívül biztosítékkal bekapcsolható **watchdog** timer is (ha bármikor túl sok ideje nem futott le kód újraindítja magát)

Valós idejű számláló: Asszinkron módon is tud futni egy számláló

SPI, **I2C/TWI:** Kommunikációs módszerek (Kijelzők, illesztő chipek, memória és más perifériákat lehet vele csatolni)

Energia szintek: Lehet altatni a chipet és valamilyen külső jelre vagy időzítőre felébreszteni. Ez kifejezetten hasznos akkumulátoros projekteknél mert egyes alvó állapotokba mikro ampre a chip fogyasztása amivel éveken keresztül tud gombelemről is futni.

AVR Timer számológép: https://eleccelerator.com/avr-timer-calculator/

Praktikák kód optimalizálásra

- **Felesleges kódrészlet:** amit lehet szervezz ki osztályokba vagy legalább változókba/függvényekbe
- **Lehető legkisebb változó megválasztása**
- **Lehető legközelebbre helyezd a változókat:** lokális jobb mint a globális
- **Stringek kiírása:** F() macróval flash-be menti el a stringeket a RAM helyett.
- **Konstansok:** hasonlóan a PROGMEM utasítással a konstans változókat is át lehet mozgatni
- **Regiszter használat:** gyorsabb i/o elérés és kevesebb függelék kód